Technical Report

UCAM-CL-TR-178
ISSN 1476-2986

Number 178

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Totally verified systems: linking verified software to verified hardware

Jeffrey J. Joyce

September 1989

ISSN 1476-2986

Contents

1 Introduction 1

2 The Compiler Correctness Problem 3

3 The Source Language 5

4 Representing Hierarchical Structure 6

5 Semantics 9

6 Compiler Overview 13

7 Compiling Expressions and Commands 15

8 Compiling Complete Programs 17

9 Assembling Intermediate Code 18

10 Combining Two Levels of Correctness Results 21

11 Summary 22

Totally Verified Systems: Linking Verified Software to Verified Hardware

Jeffrey J. Joyce
University of Cambridge

Abstract. We describe exploratory efforts to design and verify a compiler for a formally verified microprocessor as one aspect of the eventual goal of building *totally verified systems*. Together with a formal proof of correctness for the microprocessor, this yields a precise and rigorously established link between the semantics of the source language and the execution of compiled code by the fabricated microchip. We describe, in particular: (1) how the limitations of real hardware influenced this proof; and (2) how the general framework provided by higher-order logic was used to formalize the compiler correctness problem for a hierarchically structured language.

Keywords. compiler correctness, hardware verification, machine-assisted theorem proving, higher-order logic, safety-critical systems.

1. Introduction

Many safety-critical systems are implemented by a combination of hardware and software. The reliability of these systems depends not only on correct hardware and correct software, but also on the correctness of the compiler which provides the link between hardware and software levels. This paper describes exploratory efforts to design and verify a compiler for a formally verified microprocessor called 'Tamarack'. The source language is a very simple, hierarchically structured language with only a few basic constructs, e.g., expressions, assignment statements, while-loops, but this is enough to demonstrate how our approach could be applied to more realistic languages. We have used higher-order logic to formally specify this compiler and prove that it generates Tamarack machine code which executes correctly with respect to a denotational semantics for the source language.

The verification of this compiler builds upon an earlier proof of correctness showing that a transistor level model of the target machine satisfies a behavioural spec-

[1]Author's current address: Computer Laboratory, University of Cambridge, Pembroke Street, Cambridge CB2 3QG, England. After January 1, 1990: Department of Computer Science, University of British Columbia, 6356 Agricultural Road, Vancouver B.C., Canada V6T 1W5.

ification based on the semantics of the target machine instruction set [16]. The verification of both the compiler and the target machine in higher-order logic have been mechanically checked by the HOL system [13]. The HOL system has also been used to automatically generate substantial portions of these proofs.

The compiler correctness problem has a very long history beginning in the mid-1960's, but almost all of the previous work on this problem has been restricted to abstract, idealized target machines. These idealizations can include infinite word size and memory size, read-only code and symbolically addressed memory. By contrast, our target machine is not idealized hardware; indeed, an 8-bit version of the Tamarack microprocessor has been implemented as a CMOS microchip. Hence, our use of non-idealized hardware contributes to the more novel aspects of the work reported here.

Previous work has also generally relied upon specialized frameworks such as domain theory and algebraic concepts which are well-suited to the compiler correctness problem. But in the context of verifying both a compiler and the hardware of the target machine, a very general framework is needed to handle this many-sided problem. Such a framework is provided by the HOL system, a mechanization of higher-order logic, which has been used to reason about all kinds of computational systems.

Like most other examples of compiler verification, we ignore the problems of parsing and syntax analysis and use the abstract syntax of the source language as our starting point. The compiler is defined as a function which is applied to the parse tree of a program to generate code for the target machine. Semantic functions are applied in a similar way to the parse tree to generate the denotation of a program.

The work described here explores one aspect of the eventual goal of building *totally verified systems*. Assuming that our transistor level specification is an accurate model of the hardware, the compiler correctness proof combined with our earlier proof of correctness for the target machine results in a precise and rigorously established connection between the source language semantics and the execution of compiled code on the fabricated microchip. Hence, the semantics of the source language can be used to directly reason about the effect of running compiled programs on real hardware.

A detailed description of the Tamarack compiler and its formal verification is given in a separate report [17]. In this paper, we briefly outline the structure of this proof describing, in particular: (1) how the limitations of real hardware influenced this proof; and (2) how the general framework provided by higher-order logic was used to formalize the compiler correctness problem for a hierarchically structured language.

2. The Compiler Correctness Problem

The compiler correctness problem is easier to formulate than the general problem of program correctness. Unlike the general case, the compiler correctness problem has a built-in starting point for stating correctness, namely, the semantics of the source language. Intuitively, this problem is a question of whether the execution of compiled code agrees with the semantics of the source language. Compiler correctness is often expressed by the commutativity of a diagram similar to the one shown in Figure 1 where the two paths in the diagram from the source language programs to target language meanings (around opposite corners) are functionally identical.

$$\begin{tikzcd}
\text{Source Language Programs} \arrow[r, "\text{Compiler}"'] \arrow[d, "\text{Source Language Semantics}"'] & \text{Target Language Programs} \arrow[d, "\text{Target Language Semantics}"] \\
\text{Source Language Meanings} \arrow[r, "\text{Abstraction Functions}"] & \text{Target Language Meanings}
\end{tikzcd}$$

Figure 1: Compiler Correctness expressed by Commutativity.

The earliest example of compiler correctness (that we are aware of) was described more than twenty years ago by J. McCarthy and J. Painter [20]. They verified an algorithm for compiling arithmetic expressions into code for an abstract machine. This early work established a paradigm for subsequent work on compiler correctness (as summarized by A. Cohn [7]): (1) abstract syntax; (2) idealized hardware; (3) abstract specification of the compiler; (4) denotational source language semantics; (5) operational target machine semantics; (6) correctness stated as a relationship between the denotation of a program and the execution of its compiled form; and finally, (7) proofs by induction on the structure of source language expressions.

Subsequent developments include those described by: D. Kaplan [18]; R. Burstall and P. Landin [4]; R. Milner and R. Weyhrauch [23]; F. Morris [25,26]; L. Chirica [5]; R. Milne and C. Strachey [22]; J. Goguen et al. [11]; B. Russell [31]; A. Cohn [7]; W. Polak [29,30]; J. Thatcher et al. [33]; L. Chirica and D. Martin [6]; and P. Collier [9]. These developments include the use of algebraic methods and domain theory, more language features, verification by formal proof based on axioms and inference rules, mechanical assistance for proof-checking and proof-generation, and correctness proofs about parsing and syntax analysis.

However, all of the work mentioned above involves the use of a target machine with idealized features. Typically, the target machine has no finite limitations on word size or memory size. Another idealization is the use of read-only code, which avoids the problem of showing that a compiled program is not over-written during its execution. The target machine is occasionally provided with abstract mechanisms such as an infinite stack or display mechanism (admittedly, finite approximations of these mechanisms are available in real hardware). In some compiler verification examples, the memory of the target machine is addressed symbolically by program variables, dodging the problem of symbol table generation. Similarly, the target language may be block structured to avoid the complication of generating unique labels for instructions.

These idealizations, while simplifying the problem, can also be justified as reasonable strategies for structuring both the compiler and a proof of its correctness into several layers. Non-idealized aspects of hardware, in the context of programming language semantics and implementation, were recognized long ago; for instance, see papers by C. Hoare [15] and M. Newey [27]. But to our knowledge, these details and the attendant proof complexity have not been confronted until recently, in the work described here, and in J Moore's formal verification of the Piton assembler for the FM8502 microprocessor [24]. As part of the verified stack described by W. Bevier et al. [2], Piton provides considerable support as an intermediate language with stack-based instructions, typed data and recursive procedures [1]. Moore's proof takes account of the finite limitations of hardware; he also deals with issues such as allocating memory for program variables and loading compiled code and data into a single memory image. The semantics of Piton are given operationally by a formally defined interpreter expressed as a recursive function in the Lisp-like syntax of Boyer-Moore logic [3].

Our exploratory efforts with a simple 'toy' language are quite modest when compared to Moore's work on Piton. However, we have tackled a somewhat different problem by considering a hierarchically structured source language. We expect that methods similar to those described in this paper could be used to verify a compiler for a structured assembly language such as Vista [19] which is being used to write applications software for the (partially) verified Viper microprocessor [8,10,19]. Another important difference is the operational-style semantics of Piton in contrast to our denotational approach. We believe that the abstract and concise nature of a denotational semantics will be an advantage when compiler correctness results are used to relate conventional forms of reasoning about software (e.g., a verification condition generator based on Hoare logic) to the execution of compiled software on verified hardware.

[1] As another level in the verified stack described by Bevier et al. [2], W. Young has verified a code generator for a hierarchically structured source language with Piton as the target language [34].

3. The Source Language

Our source language is a very simple imperative language. It is not intended to be a useful programming language; it only provides a few basic constructs in order to demonstrate how our approach could be applied to more realistic languages. For instance, the only kind of compound arithmetic expression is a plus-expression. Conditional statements are not included because while-loops cover all the proof difficulties (and more) presented by conditional statements. We also simplify code generation by imposing an unusual restriction on plus-expressions and equal-expressions: the left-hand sides of these expressions must be atomic. An informal description of the abstract syntax for this language is shown below.

```
Aexp ::= {0,1,2,...} | Vars | Vars + Aexp
Bexp ::= Vars = Aexp | not Bexp
Cexp ::= skip | Vars := Aexp | Cexp ; Cexp | while Bexp do Cexp
```

There are three syntactic categories: arithmetic expressions, Boolean expressions and command expressions (or simply, commands). Vars is a set of string tokens which are used as variable names in programs, e.g., 'i' and 'sum' in the program shown in Figure 2. This program, called "SUM_0_to_9", computes the sum of the numbers 0 to 9 inclusive.

```
i := 0;
sum := 0;
while not (i = 10) do
  sum := sum + i;
  i := i + 1
```

Figure 2: The SUM_0_to_9 Program.

A denotational semantics for this simple language involves the definition of *semantic functions* for each syntactic category, namely, SemAexp, SemBexp and SemCexp. These functions map *syntactic objects* to their denotations as suggested by the type declarations,

$$\begin{aligned} \texttt{SemAexp}&: Aexp \to Asem \\ \texttt{SemBexp}&: Bexp \to Bsem \\ \texttt{SemCexp}&: Cexp \to Csem \end{aligned}$$

where *Aexp*, *Bexp* and *Cexp* are syntactic domains and *Asem*, *Bsem* and *Csem* are the corresponding semantic domains.

These semantic functions can be described informally by a set of *semantic clauses* using the *emphatic brackets* $[\![$ and $]\!]$ to surround syntactic objects when applying semantic functions to them [12]. *Semantic operators* on the right-hand sides of these clauses are used to construct denotations from variables, constants and denotations of sub-expressions.

```
SemAexp ⟦v⟧ = SemVar v
SemAexp ⟦c⟧ = SemConst c
SemAexp ⟦v + aexp⟧ = SemPlus (v,SemAexp ⟦aexp⟧)

SemBexp ⟦v = aexp⟧ = SemEq (v,SemAexp ⟦aexp⟧)
SemBexp ⟦not bexp⟧ = SemNot (SemBexp ⟦bexp⟧)

SemCexp ⟦skip⟧ = SemSkip
SemCexp ⟦v := aexp⟧ = SemAssign (v,SemAexp ⟦aexp⟧)
SemCexp ⟦cexp1 ; cexp2⟧ = SemThen (SemCexp ⟦cexp1⟧,SemCexp ⟦cexp2⟧)
SemCexp ⟦while bexp do cexp⟧ = SemWhile (SemBexp ⟦bexp⟧,SemCexp ⟦cexp⟧)
```

To formally define the functions SemAexp, SemBexp and SemCexp, we need a suitable representation for syntactic objects. This representation must allow SemAexp, SemBexp and SemCexp to be defined as functions which satisfy the above (sometimes recursive) semantic clauses. The next section of this paper describes how syntactic objects can be represented in logic as parse trees.

4. Representing Hierarchical Structure

Many of the specialized frameworks used in earlier work on compiler verification directly support the representation of syntactic objects. While a general framework does not necessarily provide this support, it is still possible to represent syntactic objects using only rudimentary data types. We have demonstrated how this can be done in higher-order logic using a relatively concrete model for the representation of syntactic objects as parse trees such as the one shown in Figure 3.

In a conventional programming language, a parse tree can be implemented by an indexed list of records. The structure of the tree would be represented by pointers (record indices) in each record to zero, one or two sub-expression(s). Such data structures can be modelled in higher-order logic [2] using: (1) n-tuples to represent records; and (2) functions from indices to n-tuples to represent indexed lists of records. Since the representing type does not restrict how records are structurally composed into parse trees, it is necessary to have *validity predicates*, ValidAexp, ValidBexp and ValidCexp, to check whether a parse tree conforms to the abstract syntax of the source language.

[2]The HOL formulation of higher-order logic associates a *type* with every term. Every type is a primitive type (e.g., Booleans, natural numbers, string tokens) or built up from existing types using type constructors. Cartesian product is expressed by $ty1 \times ty2$ while $ty1 \rightarrow ty2$ denotes the type of all total functions with arguments of type $ty1$ and results of $ty2$.

Based on this representation for parse trees, we can define higher-order *mapping functions*, MapAexp, MapBexp and MapCexp, which allow a set of operations to be applied to the nodes of a parse tree in the same way that the Lisp function 'mapcar' allows an operation to be applied to the elements of a list. We use these mapping functions to define operations on parse trees by specifying operations for each kind of expression. These operations are applied recursively to the entire parse tree.

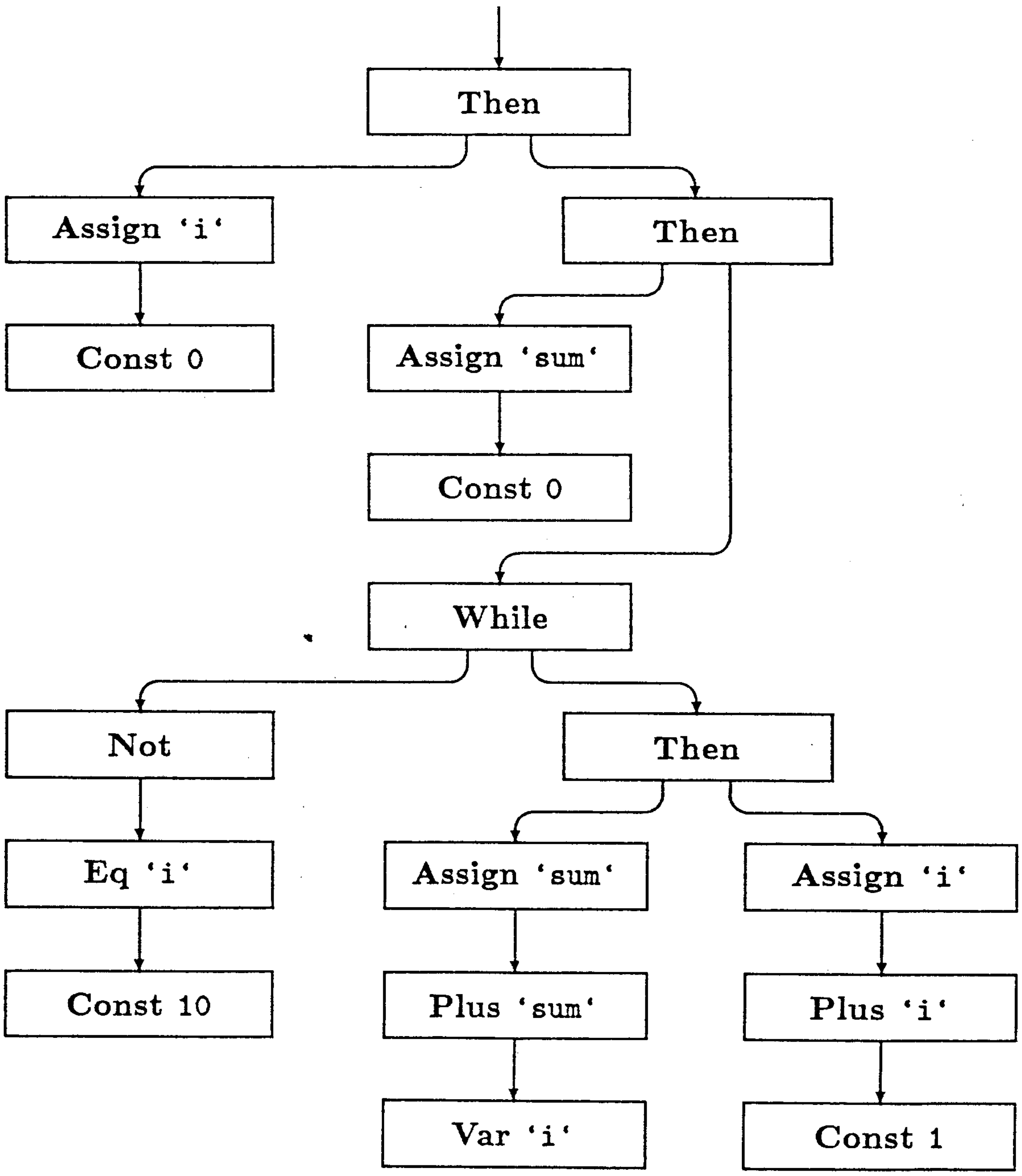

Figure 3: The Parse Tree for the SUM_0_to_9 Program.

For example, the definition of SemCexp (given in the next section) uses the mapping function MapCexp to recursively apply semantic operators to the parse tree of a command. This use of MapCexp is illustrated by the following term which denotes the result of applying SemCexp to the parse tree in Figure 3.

```
SemThen (
  SemAssign ('i',SemConst 0),
  SemThen (
    SemAssign ('sum',SemConst 0),
    SemWhile (
      SemNot (SemEq ('i',SemConst 10)),
      SemThen (
        SemAssign ('sum',SemPlus ('sum',SemVar 'i')),
        SemAssign ('i',SemPlus ('i',SemConst 1))))))
```

In addition to defining operations on expressions, we will also want to prove theorems about the result of applying such operations to expressions. To prove that a property holds for all expressions in a particular category, it is sufficient to show that the property holds for each kind of expression in the category assuming that it holds for all sub-expressions. This form of logical argument is called structural induction. Based on our representation for parse trees, we can prove structural induction theorems for each of the syntactic categories of the source language.

For instance, structural induction for arithmetic expressions is expressed by the following theorem. The predicates IsVar, IsConst and IsPlus are selectors for the three different kinds of arithmetic expressions and the function RightOf is used to obtain the sub-expression of a plus-expression.

```
⊢thm ∀P.
       (∀exp. IsVar exp ⟹ P exp) ∧
       (∀exp. IsConst exp ⟹ P exp) ∧
       (∀exp.
         IsPlus exp ∧ ValidAexp (RightOf exp) ⟹
         (P (RightOf exp) ⟹ P exp))
       ⟹
       ∀exp. ValidAexp exp ⟹ P exp
```

Structural induction only holds for valid parse trees; however, we may assume, as part of the inductive hypothesis, that the parse tree for the sub-expression is valid (in the case of a plus-expression). Structural induction theorems for Boolean and command expression have similar constraints.

The use of validity predicates to check whether a parse tree conforms to the abstract syntax of the source language is slightly cumbersome. Validity predicates provide a simple way to represent structure in a generalized framework using only rudimentary data types. A more elegant approach avoids the use of validity predicates by formally introducing new types (as sub-types of the representing type) which contain (by definition) only valid syntactic objects.

We have used a relatively concrete representation for syntactic objects as collections of records organized into parse trees. The details of this representation are unimportant and are hidden at early point in our proof by the derivation of *abstract specifications* for the mapping functions MapAexp, MapBexp and MapCexp and the derivation of the above-mentioned structural induction theorems. In a more abstract approach, the unimportant details of a concrete representation can be entirely avoided by directly introducing a recursive type whose elements are (by definition) valid syntactic objects. This approach was taken by Cohn [7] working in LCF which is also a typed logic. This more abstract approach could also be followed in our higher-order logic framework - a task made easier by T. Melham's recent implementation of a recursive data types package for the HOL system [21].

To summarize this section, syntactic objects can be represented as parse trees which, in turn, can be represented by rudimentary data types in a generalized framework such as higher-order logic. Operations on parse trees can be defined in terms of a set of mapping functions; reasoning about parse trees is supported by a corresponding set of structural induction theorems. Full details on this representation, the mapping functions and the structural induction theorems are given in [17].

5. Semantics

A denotational semantics for the source language can be defined in higher-order logic using higher-order functions and relations as the denotations of expressions and commands respectively. This is a somewhat different framework than usual, i.e., Scott's logic for computable functions, but it is denotational in the sense that program constructs are mapped to abstract mathematical entities [12]. M. Gordon has also used higher-order logic to represent a denotational semantics in a similar manner [14].

The execution of a program is modelled by a sequence of states where each state is a mapping from variable names to their values. In this simple language only natural numbers can be assigned to variables. Hence, a state is represented by a function from string tokens to the natural numbers as shown by the following type abbreviation.

$$state = tok \rightarrow num$$

The execution of a source language program results in a sequence of updates to the current state. We use a standard model from denotational semantics for the effect of an update. The function Update creates a new state identical to the current state except for the updated variable which is assigned a new value. The following definition introduces some of our notation: Update is defined in terms of a function-denoting λ-expression and a conditional expression of the form "b ⇒ t1 | t2".

$\vdash_{def}$ Update (s:*state*,x,y) = λz. (x = z) ⇒ y | (s z)

The denotations for arithmetic and Boolean expressions are functions which specify the value of the expression in terms of the current state. The denotation of a command is a relation on pairs of initial and final states. The following type abbreviations summarize the types of denotations used for each of the three syntactic categories. These denotations are each parameterized by a number, namely, the word size of the target machine.

$$\begin{aligned} Asem &= num \to state \to num \\ Bsem &= num \to state \to bool \\ Csem &= num \to (state \times state) \to bool \end{aligned}$$

We can now begin to define semantic operators for expressions and commands in the source language. The definition of SemVar states that the denotation of a variable is its value in the current state. This operator is a *curried* function which takes its arguments 'one at a time'. When SemVar is applied to the first of its arguments, i.e., SemVar v, the result is a term with the type given by the type abbreviation *Asem* (where ws is the word size of the target machine).

$\vdash_{def}$ SemVar (v:*tok*) = λws. λq. q v

The denotation of a constant is the value of the constant modulo the word size of the target machine. This use of the MOD function is due to our eventual goal of relating the semantics of the source language to the execution of compiled programs. Modular arithmetic is a convenient way of taking into account the finite word size of non-idealized hardware; an early example of this use of modular arithmetic appears in Hoare's seminal paper on axiomatic semantics [15].

$\vdash_{def}$ SemConst (c:*num*) = λws. λq. c MOD 2^{ws}

A plus-expression is an example of a compound expression; its denotation is obtained from its immediate constituents, in this case, from the sub-expression on the right-hand side of the '+'. Modular arithmetic is also used here to model the finite word size of the target machine.

$\vdash_{def}$ SemPlus (v:*tok*,s:*Asem*) = λws. λq. ((q v) + (s ws q)) MOD 2^{ws}

The semantic operator for equal-expressions is parameterized by the string token appearing on the left-hand side of the '=' and by the denotation of its arithmetic sub-expression. The semantic operator for not-expressions is parameterized by the denotation of its Boolean sub-expression.

$\vdash_{def}$ `SemEq (v:`*tok*`,s:`*Asem*`) = λws. λq. (q v) = (s ws q)`

$\vdash_{def}$ `SemNot (s:`*Bsem*`) = λws. λq. ¬(s ws q)`

The semantic operators for commands yield relations on pairs of states. The simplest case is the `Skip` command which has no effect on the state. Therefore, the initial and final states of a `Skip` command are related if they are identical [3].

$\vdash_{def}$ `SemSkip = λws. λ(q1,q2). q1 = q2`

In the case of an assignment statement, the final state is obtained from the initial state by the `Update` function.

$\vdash_{def}$ `SemAssign (v:`*tok*`,s:`*Asem*`) =`
`λws. λ(q1,q2). q2 = Update (q1,v,s ws q1)`

In defining the semantics of a then-command (two commands in sequence), the two sub-commands must share a common intermediate state. Higher-order existential quantification is used to hide this intermediate state in the definition of `SemThen`. In a more standard framework, the denotation for a sequence of commands would be obtained by the functional composition of two partial functions. Partial functions allow for the possibility of non-terminating commands; however, all functions in higher-order logic are total. For this reason, we are using relations instead of partial functions. Our use of existential quantification for the denotation of a then-command is the analogue of functional composition for relations.

$\vdash_{def}$ `SemThen (s1:`*Csem*`,s2:`*Csem*`) =`
`λws. λ(q1,q2). ∃q3. s1 ws (q1,q3) ∧ s2 ws (q3,q2)`

The function `Step` is defined (by primitive recursion) to describe the condition where *n* iterations of a while-loop result in a final state, that is, a state in which the Boolean condition is false. Zero iterations of the while-loop is equivalent to the execution of a `Skip` command; otherwise, *n* iterations of the while-loop has the same effect as executing the body of the while-loop once followed by *n*-1 iterations of the while-loop. The semantic operators `SemSkip` and `SemThen` are used to define the zero and non-zero cases respectively. Since the actual number of iterations is not relevant to the semantics of a while-loop, this number is hidden by existential quantification in the definition of `SemWhile`.

[3] Predicates (including relations) in the HOL formulation of higher-order logic are simply functions which return Boolean values. Hence, the lambda expression, `λ(q1,q2). q1 = q2` denotes the equality relation for pairs of states.

```
⊢def Step n (s1:Bsem,s2:Csem) ws (q1,q2) =
       (n = 0) ⇒ (((s1 ws q1) = F) ∧ SemSkip ws (q1,q2)) |
         (((s1 ws q1) = T) ∧
          SemThen (s2,Step (n-1) (s1,s2)) ws (q1,q2))

⊢def SemWhile (s1:Bsem,s2:Csem) =
       λws. λ(q1,q2). ∃n. Step n (s1,s2) ws (q1,q2)
```

Although our use of higher-order logic is an unusual framework for denotational semantics, some familiar properties can be derived for the semantic operators from the definitions given in this section. For instance, assuming for a moment that our source language also includes conditional statements, the while-loop ,

⌜while bexp do cexp⌝

should have the same meaning as,

⌜if bexp then (cexp ; while bexp do cexp) else skip⌝

This property is expressed formally by the theorem,

```
⊢thm ∀ s1 s2.
       SemWhile (s1,s2) =
         SemCond (s1,SemThen (s2,SemWhile (s1,s2)),SemSkip)
```

where SemCond is a semantic operator for conditional statements defined as:

```
⊢def SemCond (s1:Bsem,s2:Csem,s3:Csem) =
       λws. λ(q1,q2).
         ((s1 ws q1) = T) ⇒ (s2 ws (q1,q2)) | (s3 ws (q1,q2))
```

The operators, SemVar, SemConst, SemPlus, SemEq, SemNot, SemSkip, SemAssign, SemThen and SemWhile, describe how the denotation of an expression is obtained from its top-level form and the denotations of its sub-expressions. The denotation of a complete expression (including commands, and hence, complete programs) is obtained by using the mapping functions mentioned in Section 4 to recursively apply these operators to every node in a parse tree. From the *abstract specifications* for MapAexp, MapBexp and MapCexp given in [17], it is quite easy to show that the following definitions satisfy the semantic clauses given earlier in Section 3.

```
⊢def SemAexp = MapAexp (SemVar,SemConst,SemPlus)

⊢def SemBexp = MapBexp (SemAexp,SemEq,SemNot)

⊢def SemCexp =
       MapCexp (SemAexp,SemBexp,SemSkip,SemAssign,SemThen,SemWhile)
```

Later in this paper we will show how the mapping functions are used in a similar way to compile a complete program by recursively applying *compilation operators* to every node in a parse tree.

6. Compiler Overview

The Tamarack compiler is implemented by two phases. The original motivation for splitting the compilation process into two phases was to control the complexity of the formal proof of correctness. However, the use of an intermediate form is common practice in compiler design for more conventional reasons. For instance, it may be possible to compile more than one source language into the intermediate form and/or compile the intermediate form into the machine code of more than one target machine. This also suggests certain opportunities for re-using correctness results.

The first phase compiles the hierarchically structured program into a flat intermediate form called SM code. In general, this is a process of compiling an expression by first compiling its sub-expressions (if any) and then using the result to generate code for the expression itself. The second phase of the compiler assembles SM code into executable Tamarack machine code called TM code. To generate TM code from the intermediate form, a symbol table is constructed to map symbols in the source program to memory addresses. Each SM instruction is mapped to a fragment of TM code where each TM instruction is a 3-bit opcode and an address field packed together into a single memory word. This second phase of the compilation process performs (very simple versions of) the tasks associated with the assembler and linking loader in a conventional programming environment. The two phases of the compilation process are shown in Figure 4 where the example SUM_0_to_9 program is first compiled into SM code and then assembled into TM code.

As an intermediate form, SM code shares some common features with the source language. In both cases, storage is addressed symbolically by variable names and 'program space' is separate from data and cannot be over-written. However, SM code also shares some common features with the target language, in particular, they are both linear sequences of accumulator-based instructions.

The semantics of SM code are described operationally by the specification of an abstract machine (called an SM machine) which directly executes this intermediate form. The SM machine consists of a fixed program, an infinite, symbolically addressed store, a program counter and an accumulator. For simplicity, we have designed the SM machine to operate exclusively on natural numbers where multiple data types might otherwise have been used. The Boolean values T and F are represented by the natural numbers 0 and 1 respectively. Modular arithmetic is used to model the finite word size of the target machine.

```
i := 0;
sum := 0;
while not (i = 10) do
  sum := sum + i;
  i := i + 1
```

⇓

```
base:       (‘CONST‘,ARB,0)
            (‘ST‘,‘i‘,ARB)
            (‘CONST‘,ARB,0)
            (‘ST‘,‘sum‘,ARB)
base+4:     (‘CONST‘,ARB,10)
            (‘EQ‘,‘i‘,ARB)
            (‘NOT‘,ARB,ARB)
            (‘NOT‘,ARB,ARB)
            (‘JZR‘,ARB,base+16)
            (‘LD‘,‘i‘,ARB)
            (‘ADD‘,‘sum‘,ARB)
            (‘ST‘,‘sum‘,ARB)
            (‘CONST‘,ARB,1)
            (‘ADD‘,‘i‘,ARB)
            (‘ST‘,‘i‘,ARB)
            (‘JMP‘,ARB,base+4)
base+16:
```

⇒

```
base:       (LD n (base+2))
            (JMP n (base+3))
            (0 MOD 2^(n+3))
base+3:     (ST n (symtab ‘i‘))
base+4:     (LD n (base+6))
            (JMP n (base+7))
            (0 MOD 2^(n+3))
base+7:     (ST n (symtab ‘sum‘))
base+8:     (LD n (base+10))
            (JMP n (base+11))
            (10 MOD 2^(n+3))
base+11:    (SUB n (symtab ‘i‘))
            (JZR n (base+16))
            (LD n (base+15))
            (JMP n (base+16))
            1
base+16:    (JZR n (base+20))
            (LD n (base+19))
            (JMP n (base+23))
            0
            (LD n (base+22))
            (JMP n (base+23))
            1
base+23:    (JZR n (base+27))
            (LD n (base+26))
            (JMP n (base+30))
            0
            (LD n (base+29))
            (JMP n (base+30))
            1
base+30:    (JZR n (base+40))
base+31:    (LD n (symtab ‘i‘))
base+32:    (ADD n (symtab ‘sum‘))
base+33:    (ST n (symtab ‘sum‘))
base+34:    (LD n (base+36))
            (JMP n (base+37))
            (1 MOD 2^(n+3))
base+37:    (ADD n (symtab ‘i‘))
base+38:    (ST n (symtab ‘i‘))
base+39:    (JMP n (base+8))
base+40:
```

Figure 4.

7. Compiling Expressions and Commands

We begin to specify the compiler by defining a function for each kind of expression which compiles that expression into SM code. Each of these functions operates only on the top-level form of the expression; sub-expressions (if any) are compiled separately and the results supplied as arguments to the function. There is a close parallel between the role of these functions in compiling a hierarchically structured program and the semantic operators mentioned earlier in Section 5. For this reason, we call these functions *compilation operators.*

The intuitive sense in which the compilation operators for arithmetic and Boolean expressions are correct is fairly obvious. For instance, the compilation operator for plus-expressions is correct if and only if execution of the compiled code loads the sum of the sub-expression and the value of the program variable into the accumulator. In general, a compilation operator is correct if and only if the effect of executing the code generated for an expression or command agrees with its denotation generated by the corresponding semantic operator. In the case of an arithmetic expression, the value of the accumulator after executing the compiled code must be equal to the value given by its denotation in the current state. For a Boolean expression, the accumulator must contain either 0 or 1 depending on whether the denotation of the expression evaluates to true or false respectively.

Because commands do not necessarily terminate, the sense in which compilation operators for commands are correct is less obvious. By 'termination', we mean that the denotation of a command relates the initial state q1 to a final state q2, i.e., that there exists a final state q2.

$\vdash_{def}$ `Terminates p ws q1 = ∃q2. SemCexp p ws (q1,q2)`

Termination, in this sense, is a property of the abstract mathematical entities denoted by source language programs; the question of whether the SM machine halts when the compiled form of the program is executed is *prima facie* a different matter. For an SM machine 'to halt', means that it eventually reaches the end of the SM code.

Using these distinct notions of termination and halting, the correctness of a compilation operator for a command is expressed by separate conditions for the terminating and non-terminating cases. In the terminating case, the SM machine must halt and the final state of its store must agree with the final state given by the corresponding denotation. In the non-terminating case, the SM machine must not halt.

After formalizing these intuitive notions of correctness, we prove that the compilation operator for each kind of expression is correct with respect to the corresponding semantic operator. These correctness results are obtained by a sequence of inferences patterned on the *symbolic execution* of the compiled code for an expression. This use of the term 'symbolic execution' is purely descriptive; our proof technique is based entirely on the inference rules of higher-order logic.

This proof technique is straightforward for atomic expressions. Each step in the symbolic execution of the compiled code corresponds to the symbolic execution of a single SM instruction. A formal model of the SM machine is specified in terms of a *next state function* which is used to step through the code generated by the compilation operator for the atomic expression. After the appropriate number of steps, we show that the resulting state of the SM machine satisfies the correctness condition for this expression.

For compound expressions (including compound commands) symbolic execution involves steps corresponding to the execution of sub-expressions in addition to the execution of single SM instructions. We assume that the appropriate correctness conditions hold for the sub-expressions and use these assumptions to reason about the execution of each sub-expression as single steps in the symbolic execution of the compound expression. The remaining steps (steps corresponding to single SM instructions) are symbolically executed by an application of the next state function.

For example, the following theorem states that the top-level form of a plus-expression is compiled correctly by the compilation operator `CmpPlus` with respect to the denotation produced by the semantic operator `SemPlus`. The correctness condition for arithmetic expressions is expressed by the predicate `AexpCorrect`. The variables `c` and `s` are the compiled code and denotation respectively of the sub-expression on the right-hand side of the '+'.

```
⊢thm ∀ c s v.
       AexpCorrect (c,s) ⟹
       AexpCorrect (CmpPlus (v,c),SemPlus (v,s))
```

Similar results are obtained for every other kind of expression in the source language. For most expressions, symbolic execution corresponds to a fixed sequence of steps. However, correctness results for while-loops are more difficult and involve proofs by mathematical induction. The terminating case for while-loops is proved by mathematical induction on the number of iterations. The non-terminating case is even more difficult because there is more than one way that a while-loop can fail to terminate: at any point, the body of the while-loop may fail to terminate, or else the while-loop itself may continue to loop forever.

There are two essential ideas being used here to reason about compound expressions. One is the idea of using assumptions about the correctness of sub-expressions to prove correctness results for compound expressions. The other is the idea of 'mixed-mode' symbolic execution where single steps correspond to either single SM instructions or to sub-expressions.

8. Compiling Complete Programs

In section 4 we showed how semantic functions for each syntactic category can be defined by applying the mapping functions MapAexp, MapBexp and MapCexp to the semantic operators. In a similar manner, *compilation functions* for each syntactic category can be obtained by applying the mapping functions to the compilation operators.

$\vdash_{def}$ CmpAexp = MapAexp (CmpVar,CmpConst,CmpPlus)

$\vdash_{def}$ CmpBexp = MapBexp (CmpAexp,CmpEq,CmpNot)

```
⊢def CmpCexp =
    MapCexp (CmpAexp,CmpBexp,CmpSkip,CmpAssign,CmpThen,CmpWhile)
```

The correctness of these compilation functions is easily established from correctness results for each compilation operator using the structural induction theorems mentioned in Section 4.

These correctness results lead directly to the following theorem where the variable p denotes any source language program. The predicate SMHalts is defined directly in terms of the formally specified model of the SM machine. For a given SM program, SMHalts describes a relation on pairs of states (q1,q2) where the SM machine begins execution in state q1 and eventually halts in state q2. Hence, SMHalts is a semantic function for SM code.

$\vdash_{thm}$ ∀p. ValidCexp p $\Longrightarrow$ (SemCexp p = SMHalts (CmpCexp p))

This theorem is the main result from the first part of our compiler correctness proof: it relates the denotational semantics of our source language to an operational semantics given by SMHalts applied to the compiled code generated by CmpCexp. We are using the term 'operational semantics' in a somewhat old-fashioned sense [4] where the semantics is given by an abstract machine and a translation from the source language into code for the abstract machine [32].

This result can also be expressed by the commutative diagram in Figure 5 which is similar to diagrams found in other discussions of the compiler correctness problem. In this case, there is no need for an abstraction function from source language meanings to target language meanings since they are identical. Consequently, the diagram has only three sides.

[4] A more recent form of operational semantics known as *Plotkin-style* or *natural* semantics has both structure and some denotational-style features [28].

$$\begin{tikzcd}
\text{Source Language Programs} \arrow[rr, "\texttt{CmpCexp}"] \arrow[dr, "\texttt{SemCexp}"'] & & \text{SM Machine Programs} \arrow[dl, "\texttt{SMHalts}"] \\
& \text{Program Meanings} &
\end{tikzcd}$$

Figure 5: Compiler Correctness expressed by Commutativity.

The second part of our correctness proof considers the assembly of SM programs into TM code establishing a correspondence between the direct execution of an SM program and the execution of an assembled SM program by the target machine. Later, this result is combined with the above theorem to obtain a direct correspondence between the denotation of a source language program and the execution of its compiled form by the target machine.

9. Assembling Intermediate Code

The external architecture of the Tamarack microprocessor consists of three state components: the memory, program counter and accumulator. A single instruction word format is used by all Tamarack instructions: a 3-bit opcode followed by n address bits. The actual size of the address field is given by a parameter throughout the formal proof of correctness. The transistor level model of the Tamarack implementation is also parameterized by the size of the address field. The correctness of this implementation has been established for all possible sizes.

The assembly of SM code into TM code requires the generation of a *symbol table*, `symtab`, which maps string tokens appearing in SM instructions to memory addresses. Symbols (i.e., string tokens) are only added to the table when they appear on the left-hand side of an assignment statement in the source language program. Since each assignment statement corresponds to an `ST` instruction in the resulting SM code, symbols are only added to the table when they appear in the SM code as an operand in an `ST` instruction. The symbol table is generated by a single pass over the SM program. When a new symbol is added to the table, it is assigned the address of the next available location in the data area of memory.

The assembly of SM code into TM code also requires an *address table*, `addrof`, which maps locations in the SM code to corresponding locations in the TM code.

The location of the TM code generated for a particular SM instruction can be determined by adding up the sizes of the code fragments generated for all preceding SM instructions. This table can also be generated by a single pass over the SM program.

The symbol table, `symtab`, and address table, `addrof`, are used in a third pass over the SM program to generate the TM code. Most SM instructions are assembled into a single TM instruction. However, a `CONST` instruction, used to load a constant into the accumulator, is assembled into a three word fragment of TM code: an `LD` instruction; a `JMP` instruction; and the constant itself which is stored in a separate memory word (i.e., as an immediate constant). The `JMP` instruction prevents the constant from being executed as an instruction. The SM instructions `EQ` and `NOT` are also assembled into multiple words of TM code. This is because `0` and `1`, representing true and false respectively, are stored as immediate constants [5].

For conceptual clarity, we have separated the assembly of SM code into three successive passes. However, there are well-known techniques, e.g., 'back-patching', which can be used to reduce the number of passes in a compiler [1].

Each of the three passes used to assemble SM code into TM code can be formally defined as an operation applied iteratively to a sequence of SM instructions; in concrete terms these functions can be defined by primitive recursion on the size of the SM code. As one might expect, correctness results for each of these passes over the SM code will involve proofs by induction on the size of the SM code.

Correctness results for symbol table generation show that the iteratively generated symbol table has several properties needed to prove that SM code is correctly assembled into TM code. For instance, we show that different symbols are mapped to different addresses. Several other less obvious properties are described in [17].

The rest of the proof is concerned with showing that SM code is correctly assembled into TM code. Intuitively, it is fairly obvious what conditions need to be satisfied: execution of the TM code must correspond to the execution of the SM program. There are several provisos, most of which arise from limitations of the finite word size and finite memory size of the target machine.

Earlier steps in the correctness proof have already been influenced by the finite limitations of the target machine: the finite word size of the target machine is a feature of both the denotational semantics of the source language and the operational semantics of SM code. However, correctness results for the first compiler phase place no bounds on the size of the SM code or the size of the store. Therefore, finite limitations of the target machine are more important in the second part of the

[5]The use of immediate constants was slightly easier (in the initial effort of developing this proof) than the more economical approach of storing a single instance of these constants in memory.

correctness proof when showing that SM code is correctly assembled into TM code. The size of addressable memory is limited by the number of bits in the address field of a target machine instruction. The memory area reserved for code must be large enough to accommodate the code generated by the assembler. Similarly, the area reserved for data must provide a separate memory word for each symbol in the symbol table. These two areas of memory must not overlap and cannot exceed the boundaries of addressable memory. We assume explicitly that these conditions are satisfied in proving that SM code is correctly assembled into TM code.

The sense in which the execution of TM code 'corresponds' to the execution of an SM program is, roughly speaking, the condition that updates to the memory state, program counter and accumulator of the target machine correspond to updates to the store, program counter and accumulator of the SM machine. There are three distinct steps in proving that execution of the assembled form of an SM program corresponds to its direct execution by the SM machine. These three steps are very briefly summarized in the next few paragraphs.

The first step establishes that the execution of the compiled form of individual SM instructions corresponds to their direct execution by the SM machine. This step in the proof is concerned with the fragments of TM code generated for each SM instruction. For each SM instruction, we prove that the symbolic execution of the TM code fragment by repeated applications of the next state function for the target machine corresponds to a single application of the next state function for the SM machine. This step also proves that execution of the code fragment does not over-write any part of the TM code.

The second step establishes that the fragments of TM code generated for each SM instruction are correctly composed into a single fragment of TM code for the entire SM program. This step is proved by mathematical induction on the size of the SM program.

The third step establishes that the execution of an assembled SM program corresponds to its direct execution by the SM machine for any number of execution steps (within the limitations of the target machine). This step is proved by mathematical induction on the number of execution steps.

The correctness result obtained from these three steps states precise details about the relationship between the execution of an assembled SM program and its direct execution by the SM machine. In very simple terms, there exists an SM machine which provides an abstract model of the target machine while executing the compiled SM program. Therefore, true statements about the direct execution of the SM program are also true statements about the execution of its compiled form by the target machine. This theorem is used in combination with earlier results to obtain a correctness result for the complete compilation process.

10. Combining Two Levels of Correctness Results

The final step in the verification of the Tamarack compiler combines correctness results for the two phases of the compilation process.

Earlier correctness results for the first compiler phase established that direct execution of the SM code generated from a terminating source language program will result in a final state which agrees with its denotation. In the case of a non-terminating program, the SM machine will not halt. For the second compiler phase, we have just seen that 'true statements about the direct execution of the SM program are also true statements about the execution of its assembled form by the target machine'.

The combination of these two results implies that a terminating source language program will be compiled into target machine code which will execute to completion and yield a final state which agrees with its denotation. This depends, of course, on whether the compiled program can be loaded into addressable memory. A precise statement of this result uses the symbol table generated by the compiler for this program to relate memory states of the target machine to the denotation of the source language program. In the case of a non-terminating program, the target machine will never complete execution of the compiled code. The correctness theorem for the terminating case is shown below.

```
⊢thm ∀ p n mem.
       ValidCexp p ∧
       CompiledAndLoaded n p (mem 0) ∧
       Terminates p (n+3) ((mem 0)∘(SymTab p))
       ⟹
       ∀ pc acc.
         TM n (mem,pc,acc) ∧
         (pc 0 = 0)
         ⟹
         ∃t.
           FirstReaches (pc,t,EndOfProg p) ∧
           SemCexp p (n+3) ((mem 0)∘(SymTab p),(mem t)∘(SymTab p))
```

To paraphrase this theorem: if the compiled code for a syntactically valid, terminating program is loaded into memory at location 0 and executed by the target machine (whose behaviour is given by the predicate TM) beginning at time 0, then the target machine will eventually reach the end of the code at some time t. When execution of this code is completed, an abstract view of the initial memory state will be related to an abstract view of the final memory state by the denotation of the program. An 'abstract view' of the memory state is obtained by using the symbol table to access the contents of the target machine memory; in the above theorem, this is expressed by use of the operator '∘' which denotes functional composition.

11. Summary

Our main correctness theorem provides a direct link between the semantics of the source language and the behavioural specification of the Tamarack microprocessor. When coupled with an earlier proof of correctness relating this behavioural specification to a transistor level model of the hardware, we obtain a precise and rigorously established connection between the denotation of a source language program and the effect of executing its compiled form on actual hardware.

A link between software and hardware levels provides a sound basis for using the semantics of the source language to reason about programs. In related work, Gordon [14] shows how Hoare logic can be embedded in higher-order logic by regarding the syntax of Hoare formulae as abbreviations for higher-order logic formulae. The axioms and inference rules of Hoare logic are then derived from semantic operators similar to the semantic operators defined in Section 5 of this paper. This means that theorems proved in Hoare logic using these axioms and rules are logical consequences of the underlying denotational semantics.

To relate this work to our correctness results for the Tamarack compiler, we would need to slightly re-formulate the axioms and rules of Hoare logic to take account of the finite size of memory words as we have done for the semantic operators in Section 5. It would then follow that theorems proved in Hoare logic about a particular program are true statements about the result of executing the compiled program on the fabricated microchip. This depends, of course, on both explicit conditions, e.g., whether the compiled code fits into addressable memory, and implicit assumptions, e.g., that the transistor level specification is an accurate model of the hardware.

In this exploratory effort, we have not ventured beyond a traditional view of formal semantics that the meaning of a program is either a partial function from initial states to final states or, as in our approach, a relation between initial and final states. However, we are interested in embedded systems where a 'batch processing' view of program behaviour is not entirely appropriate. These systems continuously interact with an environment; they are typically implemented by a fixed program compiled and loaded into the memory of one or more microprocessors. Unlike a batch job, execution of the compiled code is meant to execute forever, or at least, until the microprocessor is reset or switched off. Instead of a final outcome, we are interested in the on-going behaviour of the microprocessor while executing the compiled code. We are concerned, for instance, that the system responds correctly to external stimuli or that certain invariants are maintained. Therefore, an important direction of future work will be to investigate the relationship between suitable kinds of semantics for proving the correctness of a compiler and formalisms which can be used to reason about continuously-operating systems.

Acknowledgements

I am grateful to Juanito Camilleri, Avra Cohn, Mike Gordon, John Herbert, Miriam Leeser and Glynn Winskel who have helped in various ways with this work. This research has been funded by the Cambridge Commonwealth Trust, the Canada Centennial Scholarship Fund, the Government of Alberta Heritage Fund, the Natural Sciences and Engineering Research Council of Canada, Pembroke College, and the UK Overseas Research Student Awards Scheme.

References

[1] Alfred V. Aho and Jeffrey D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, MA., 1977.

[2] William R. Bevier, Warren A. Hunt, Jr., and William D. Young, in: Towards Verified Execution Environments, in: Procs. of the 1987 IEEE Symposium on Security and Privacy, 27-29 April 1987, Oakland, California Computer Society Press, Washington, D.C., 1987 pp. 106-115. Also Technical Report No. 5, Computational Logic, Inc., Austin, Texas, February 1987.

[3] R.S. Boyer and J S. Moore, A Computational Logic, Academic Press, New York, 1979.

[4] R.M. Burstall and P.J. Landin, Programs and their Proofs: an Algebraic Approach, in: B. Meltzer and D. Mitchie, eds., Machine Intelligence, Vol. 4, Edinburgh Univ. Press, Edinburgh, Scotland, 1969. pp. 17-43.

[5] L.M. Chirica, Contributions to Compiler Correctness, Ph.D. Thesis, Report UCLA-ENG-7697, Computer Science Dept., Univ. of California, Los Angeles, October 1976.

[6] Laurian M. Chirica and David F. Martin, Toward Compiler Implementation Correctness Proofs, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, pp. 185-214.

[7] Avra Jean Cohn, Machine Assisted Proofs of Recursion Implementation, Ph.D. Thesis, Technical Report CST-6-79, Dept. of Computer Science, Univ. of Edinburgh, April 1980.

[8] Avra Cohn, Correctness Properties of the Viper Block Model: The Second Level, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, New York, 1989, pp. 1-91. Also Report No. 134, Computer Lab., Cambridge Univ., May 1988.

[9] P.A. Collier, Simple Compiler Correctness - A Tutorial on the Algebraic Approach, Australian Computer Journ., Vol. 18, No. 3, August 1986, pp. 128-135.

[10] W.J. Cullyer, High Integrity Computing, in: M. Joseph, ed., Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, No. 331, Springer-Verlag, Berlin, 1988. pp. 1-35.

[11] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright, Initial Algebra Semantics and Continuous Algebra, Journ. of the ACM, Vol. 24, No. 1, January 1977, pp. 68-95.

[12] Michael J. C. Gordon, The Denotational Description of Programming Languages, Spring-Verlag, Berlin, 1979.

[13] Mike Gordon, A Proof Generating System for Higher-Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, Boston, 1988, pp. 73-128. Also Report No. 103, Computer Lab., Cambridge Univ., January 1987.

[14] Michael J. C. Gordon, Mechanizing Programming Logics in Higher Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, New York, 1989, pp. 387-439. Also Report No. 145, Computer Lab., Cambridge Univ., September 1988.

[15] C.A.R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol. 12, No. 10, October 1969, pp. 576-583.

[16] Jeffrey J. Joyce, Formal Specification and Verification of Microprocessor Systems, in: S. Winter and H. Schumny, eds., Euromicro 88, Procs. of the 14th Symposium on Microprocessing and Microprogramming, Zurich, Switzerland, 29 August - 1 September, 1988, North-Holland, Amsterdam, 1988, pp. 371-378. Also Report No. 147, Computer Lab., Cambridge Univ., September 1988.

[17] Jeffrey J. Joyce, A Verified Compiler for a Verified Microprocessor, Report No. 167, Computer Lab., Cambridge Univ., March 1989.

[18] Donald M. Kaplan, Correctness of a Compiler for Algol-like Programs, Stanford Artificial Intelligence Memo No. 48, Stanford Univ., July 1967.

[19] J. Kershaw, The VIPER Microprocessor, Report No. 87014, RSRE, Malvern, UK Ministry of Defence, November 1987.

[20] J. McCarthy and J. Painter, Correctness of a Compiler for Arithmetic Expressions, in: J. Schwartz, ed., Procs. of a Symposia on Applied Mathematics, American Mathematical Society, 1967, pp. 33-41.

[21] Thomas F. Melham, Automating Recursive Type Definitions in Higher Order Logic, in: G. Birtwistle and P. Subrahmanyam, eds., Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, New York, 1989, pp. 341-386. Also Report No. 146, Computer Lab., Cambridge Univ., September 1988.

[22] Robert Milne and Christopher Strachey, A Theory of Programming Language Semantics, Chapman and Hall, London, 1976.

[23] R. Milner and R. Weyhrauch, Proving Compiler Correctness in a Mechanized Logic, in: B. Meltzer and D. Mitchie, eds., Machine Intelligence, Vol. 7, Edinburgh Univ. Press, Edinburgh, Scotland, 1972, pp. 51-70.

[24] J Strother Moore, A Mechanically Verified Language Implementation, Report No. 30, Computational Logic Inc., Austin, Texas, September 1988.

[25] F. Lockwood Morris, Correctness of Translations of Programming Languages, Ph.D. Thesis, Report STAN-CS-72-303, Computer Science Dept., Stanford Univ., August 1972.

[26] F. Lockwood Morris, Advice on Structuring Compilers and Proving Them Correct, in: Procs. of the ACM Symposium on Principles of Programming Languages, Boston, Mass., October 1973, pp. 144-152.

[27] Malcolm C. Newey, Proving Properties of Assembly Language Programs, in: B. Gilchrist, ed., Information Processing 77, North Holland, 1977, pp. 795-799.

[28] Gordon D. Plotkin, A Structured Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Dept., Aarhus Univ., September 1981.

[29] Wolfgang Heinz Polak, Theory of Compiler Specification and Verification, Ph.D. Thesis, Report No. STAN-CS-80-802, Dept. of Computer Science, Stanford Univ., May 1980.

[30] Wolfgang Heinz Polak, Compiler Specification and Verification, Lecture Notes in Computer Science, No. 124, Springer-Verlag, Berlin, 1981.

[31] Bruce D. Russell, Implementation Correctness involving a Language with goto Statements, SIAM Journ. of Computing, Vol. 6, No. 3, September 1977, pp. 403-415.

[32] Joseph E. Stoy, The Scott-Strachey Approach to Programming Language Theory, The MIT Press, Cambridge MA., 1977.

[33] J.W. Thatcher, E.G. Wagner, and J.B. Wright, More on Advice on Structuring Compilers and Proving Them Correct, Theoretical Computer Science, Vol. 15, September 1981, pp. 223-245.

[34] William D. Young, A Verified Code Generator for a Subset of Gypsy, Report No. 33, Computational Logic Inc., Austin, Texas, October 1988.